Connect Four

Introduction to Artificial Intelligence Assignment II

Team Members

Name	ID
Amin Mohamed Amin Elsayed	21010310
Omar Al-Dawy Ibrahim Al-Dawy	21010864
Andrew Safwat Fawzy	21010314

How to Play ?

Connect 4 is a strategic two-player game played on a vertical grid. Each player selects a color (typically red or yellow) and takes turns dropping their colored discs into one of the seven columns from the top. The discs fall straight down, occupying the lowest available space in the chosen column.

The objective is to be the first to align four of your discs consecutively in a row, either **vertically**, **horizontally**, or **diagonally**. Players alternate turns until one player successfully connects four discs or the board becomes full.

How to Play ?

At the end of the game, if no player has connected four, the winner is determined by who has the **greater number** of completed "connect-fours." If both players have an **equal number**, the game ends in a **draw**.

Why Can We Solve Connect 4 Using Minimax, Alpha-Beta Pruning, and Expectiminimax?

Connect 4 is a zero-sum game, making it well-suited for decision-making algorithms like **Minimax**, **Alpha-Beta Pruning**, and **Expectiminimax**.

Each of these algorithms is designed to evaluate possible game states and determine the best move based on the current board configuration.

Below is an explanation of why these methods are applicable to Connect 4 and how they contribute to the AI's performance.

Why Minimax is Suitable?

Zero-Sum Game: In Connect 4, one player's gain is the other player's loss, making the game perfectly suited for Minimax.

Perfect Information: Both players have full visibility of the board, and there is no hidden information or randomness involved.

Recursive Nature: Minimax recursively explores the game tree, simulating both the maximizing player (AI) and the minimizing player (human).

Limitations of Minimax

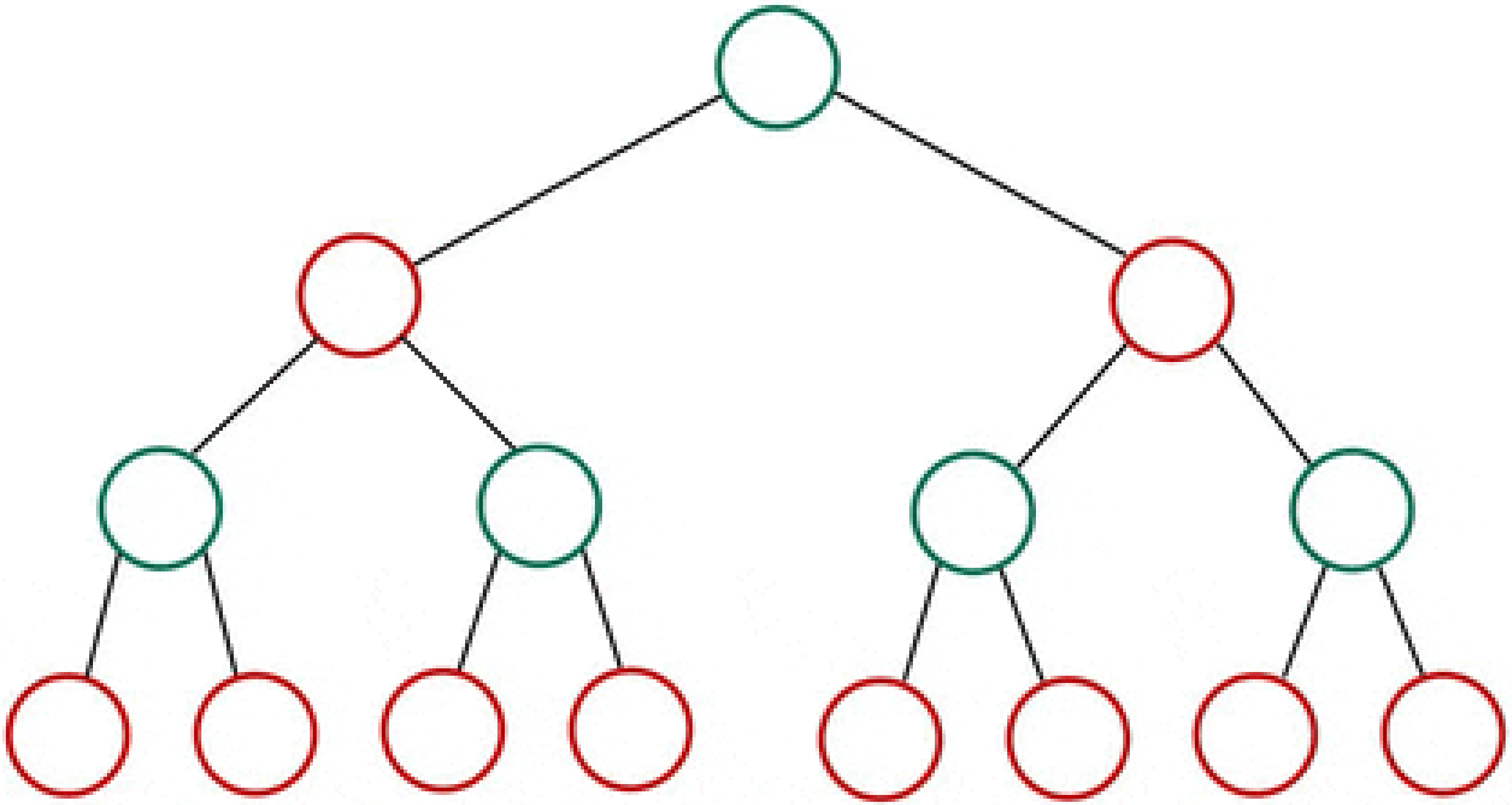
The **exponential growth** of the game tree makes it computationally expensive to explore the entire tree, especially in later stages of the game.

Without optimization, Minimax may be too slow for real-time gameplay.

Why Alpha Beta Minimax is Suitable?

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm. It eliminates branches of the game tree that do not need to be explored because they cannot influence the final decision.

- **Reduces Computational Overhead:** By "pruning" irrelevant branches, Alpha-Beta significantly reduces the number of nodes that need to be evaluated.
- **Maintains Optimality:** The pruning process does not affect the final decision, meaning the algorithm still returns the same optimal move as Minimax but in less time.
- **Depth Limitation:** It allows the AI to search deeper within the same time constraints, improving its ability to plan multiple moves ahead.



Why ExpectiMinimax is Suitable?

Expectiminimax is a variation of Minimax designed for games that involve **stochastic elements**. While Connect 4 is a deterministic game, we can apply Expectiminimax if we introduce **random or probabilistic elements** such as:

- **Randomizing** the opponent's moves based on probabilities (e.g., simulating imperfect human players).
- **Simulates Uncertainty**: In scenarios where the human opponent's behavior is uncertain, Expectiminimax can model the probabilities of different actions.

Limitations of ExpectiMinimax

- Connect 4 is **inherently deterministic**, so Expectiminimax may **not be necessary in standard gameplay** unless randomness is intentionally introduced.
- Expectiminimax is **computationally more expensive than Minimax**, as it evaluates expected values rather than deterministic outcomes.

Why Do We Need a Heuristic in Connect 4?

The **game tree for Connect 4** is exceptionally large, with a complexity of approximately **$O(10^{35})$** possible game states. This vast number makes it computationally infeasible to fully explore all possible moves and outcomes until the terminal state (i.e., when the game ends in a win, loss, or draw).

Therefore, it is essential to implement a **heuristic function** to evaluate the board state and guide the AI's decision-making process.

Why Do We Need a Heuristic in Connect 4?

A heuristic function serves as an **approximate evaluation** of a game state without traversing the entire game tree.

It provides a numerical value that indicates how favorable a particular board configuration is for the computer.

By using this function, the AI can focus on the most promising moves while ignoring less favorable branches, significantly reducing the number of states it needs to explore.

Benefits of Using a Heuristic Function

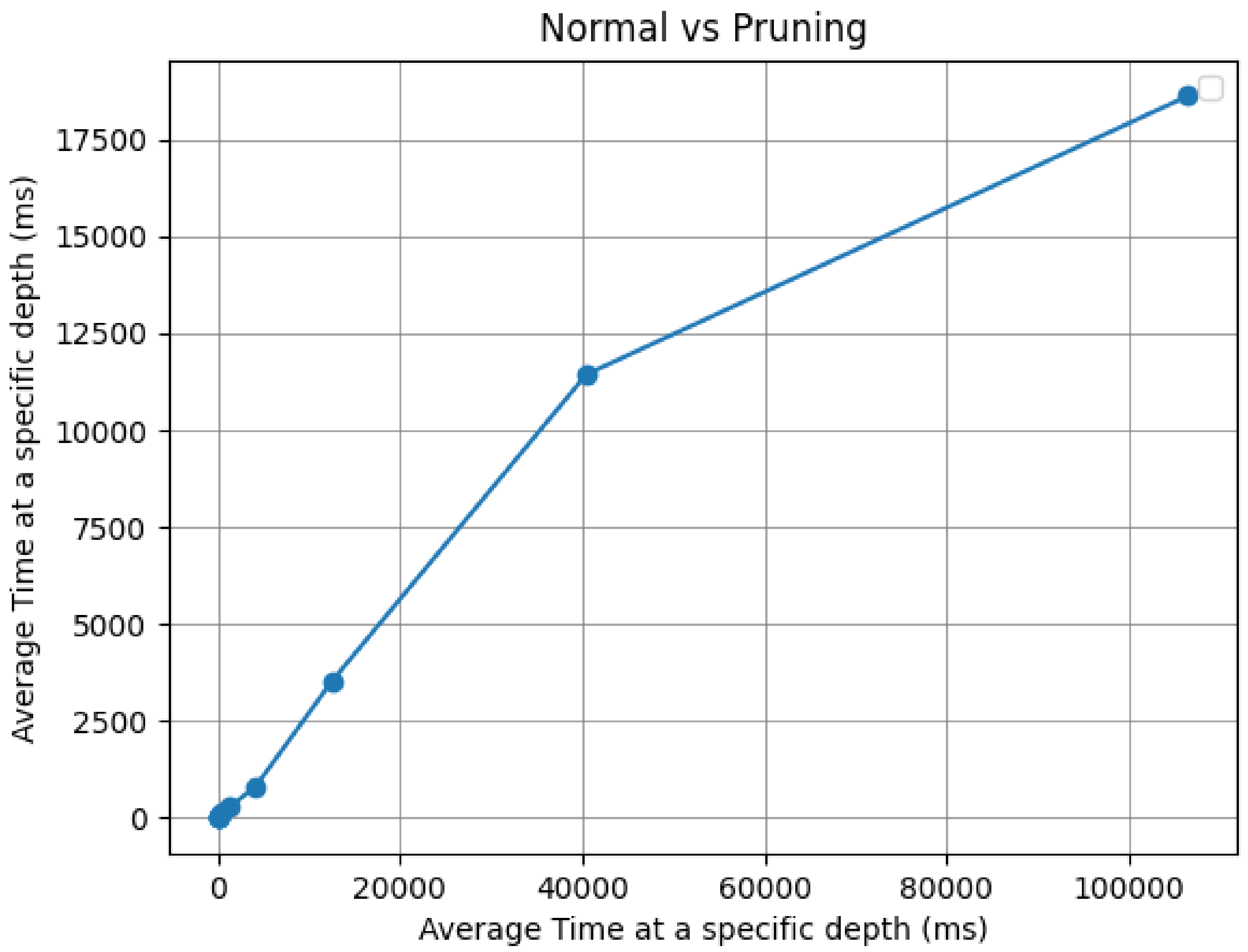
- **Reduces Computational Complexity**
  - Instead of searching the entire game tree, the AI can **truncate the tree after a fixed number of levels (K)** and evaluate the board states using the heuristic. This allows the AI to make decisions more efficiently within a reasonable timeframe.
- **Improves Decision Quality**
  - A well-designed heuristic can approximate the likelihood of winning or losing based on various board factors, such as:
  - Number of consecutive discs (2-in-a-row, 3-in-a-row, etc.).
  - Potential threats or opportunities to create a "Connect 4."
  - Blocking moves that prevent the opponent from winning.

Benefits of Using a Heuristic Function

- **Balances Between Speed and Accuracy:**
  - Since traversing the entire tree is impractical, the heuristic provides a **balance between speed and decision accuracy**. A good heuristic enables the AI to make competitive moves while keeping the computation manageable.
- **Facilitates Early Game and Mid-Game Decisions**:
  - In the early and mid-game phases, where terminal states are far away, a heuristic function is critical. It helps the AI **evaluate intermediate states** and prioritize moves that lead to long-term advantages.

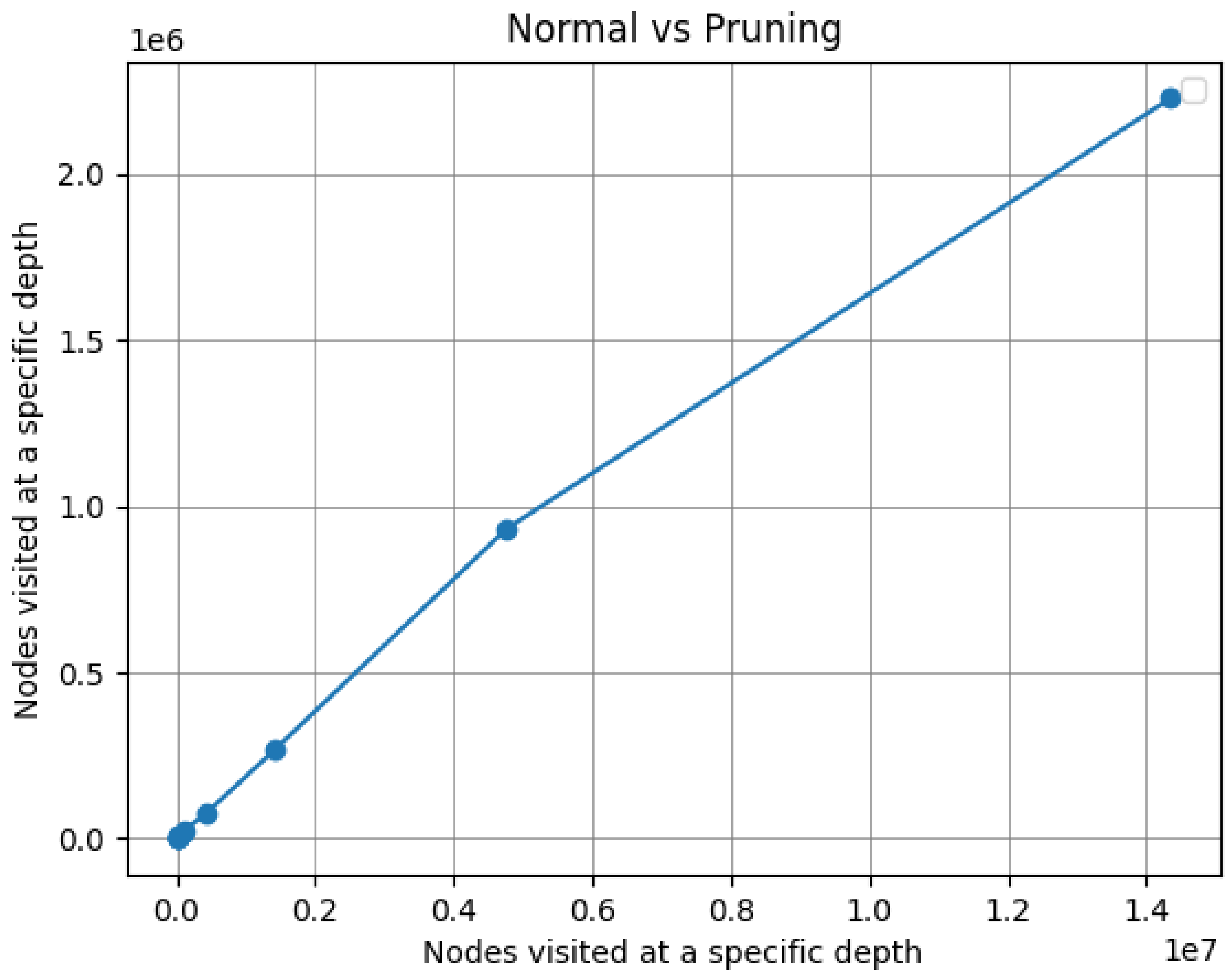
Our Heuristic Function

- We calculate some board features such as **how many fours have been connected**, **three pieces could be four**, **two pieces could be four** and **the position of the column the single pieces are distributed on**.
- Given score for each situation reflecting its revenue for the player and get the **difference** between its opponent revenue to be get the total revenue for the player we want to find the move with maximum profit for him.



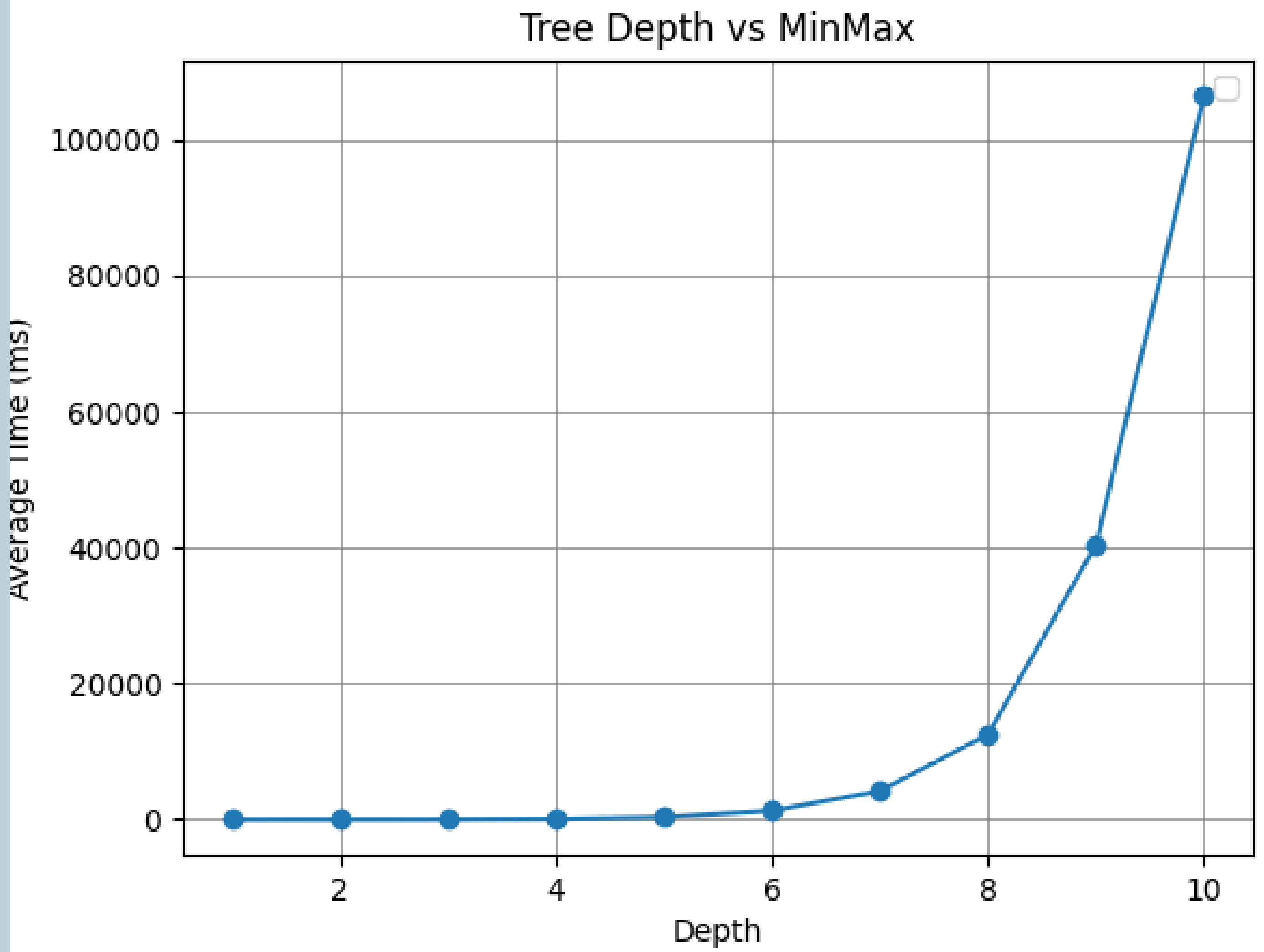
Performance

- Y-Axis: AlphaBeta Time
- X-Axis: Minmax Time



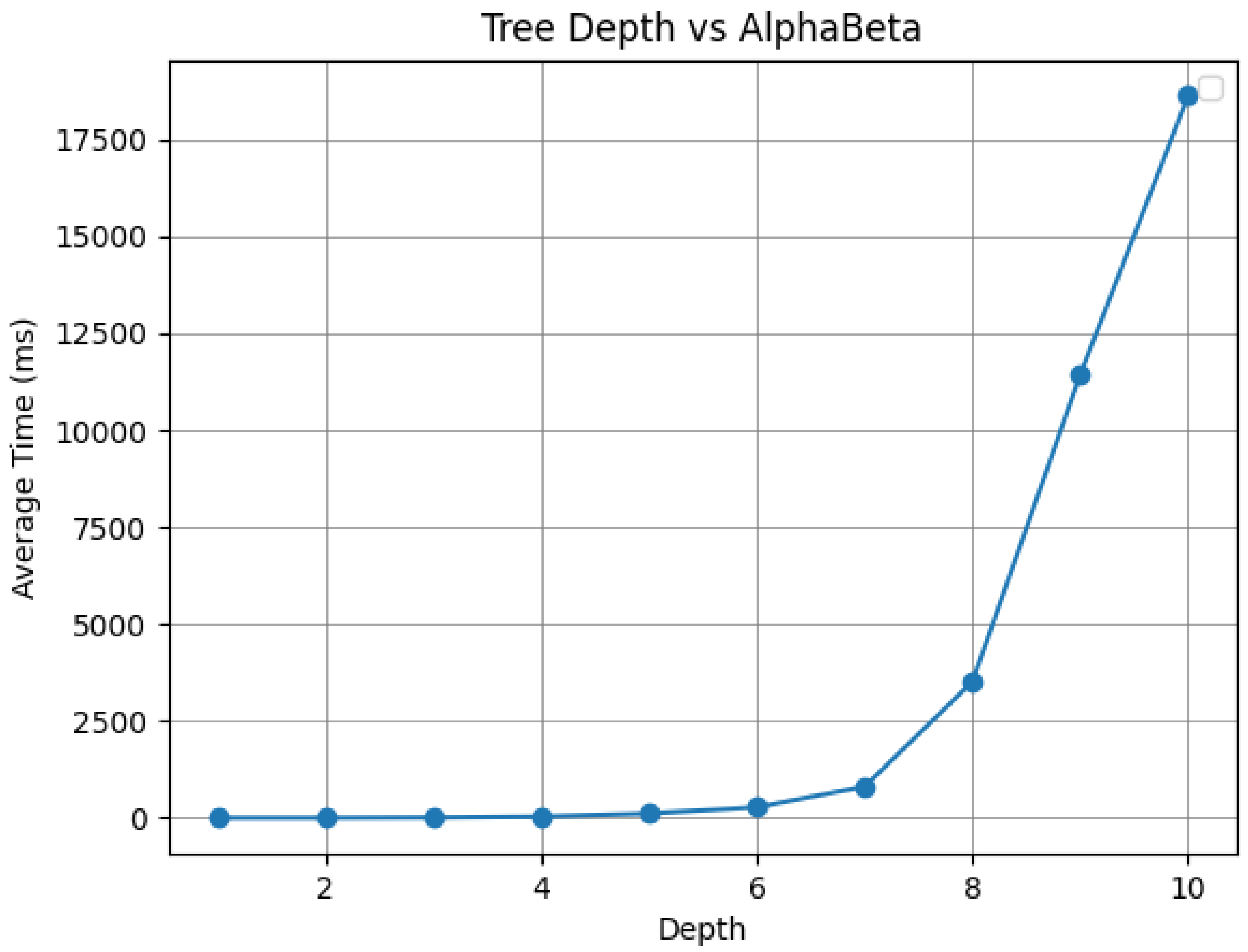
Performance

- Y-Axis: AlphaBeta Explored Nodes
- X-Axis: Minmax Explored Nodes



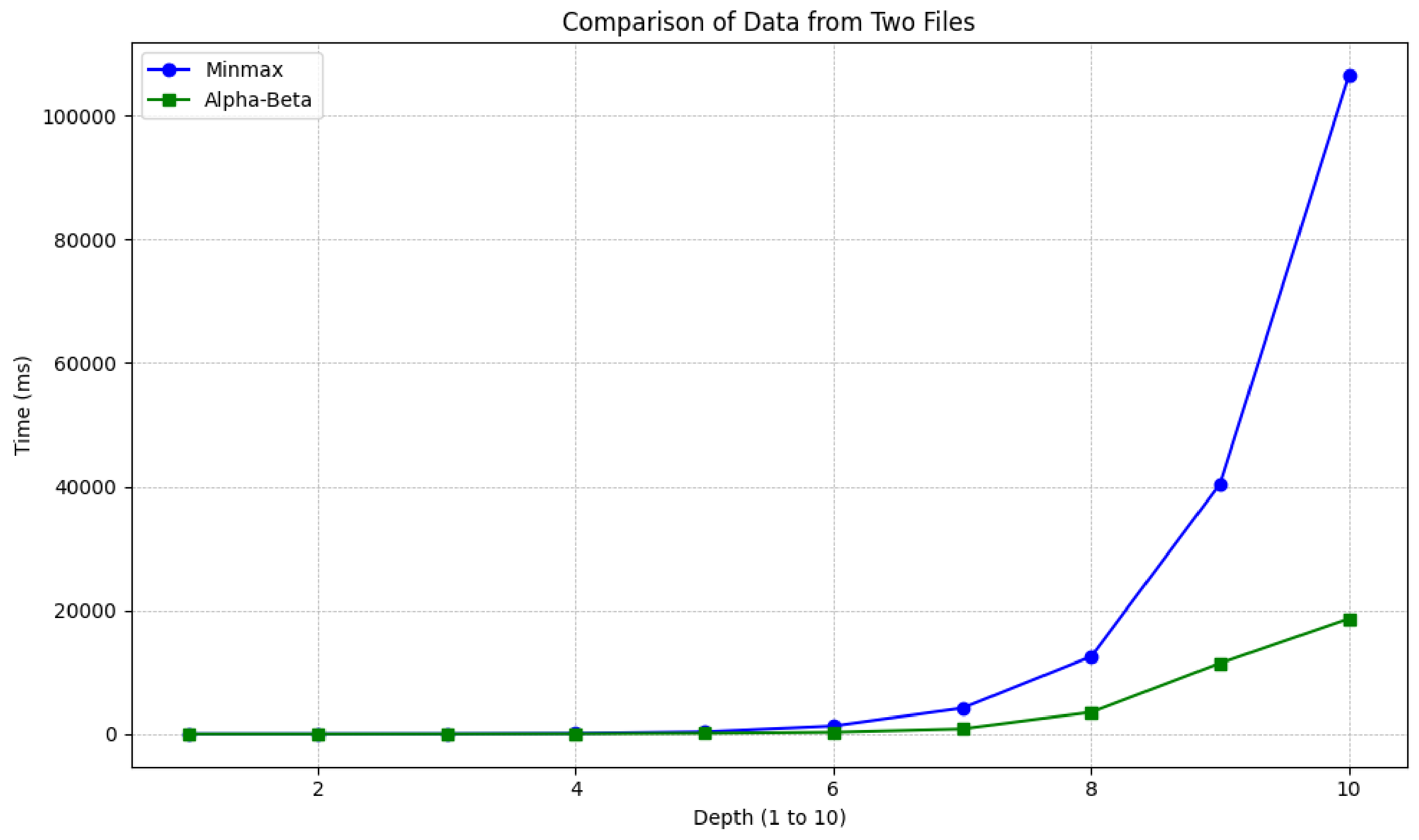
Performance

- Y-Axis: Minmax Time
- X-Axis: Depth



Performance

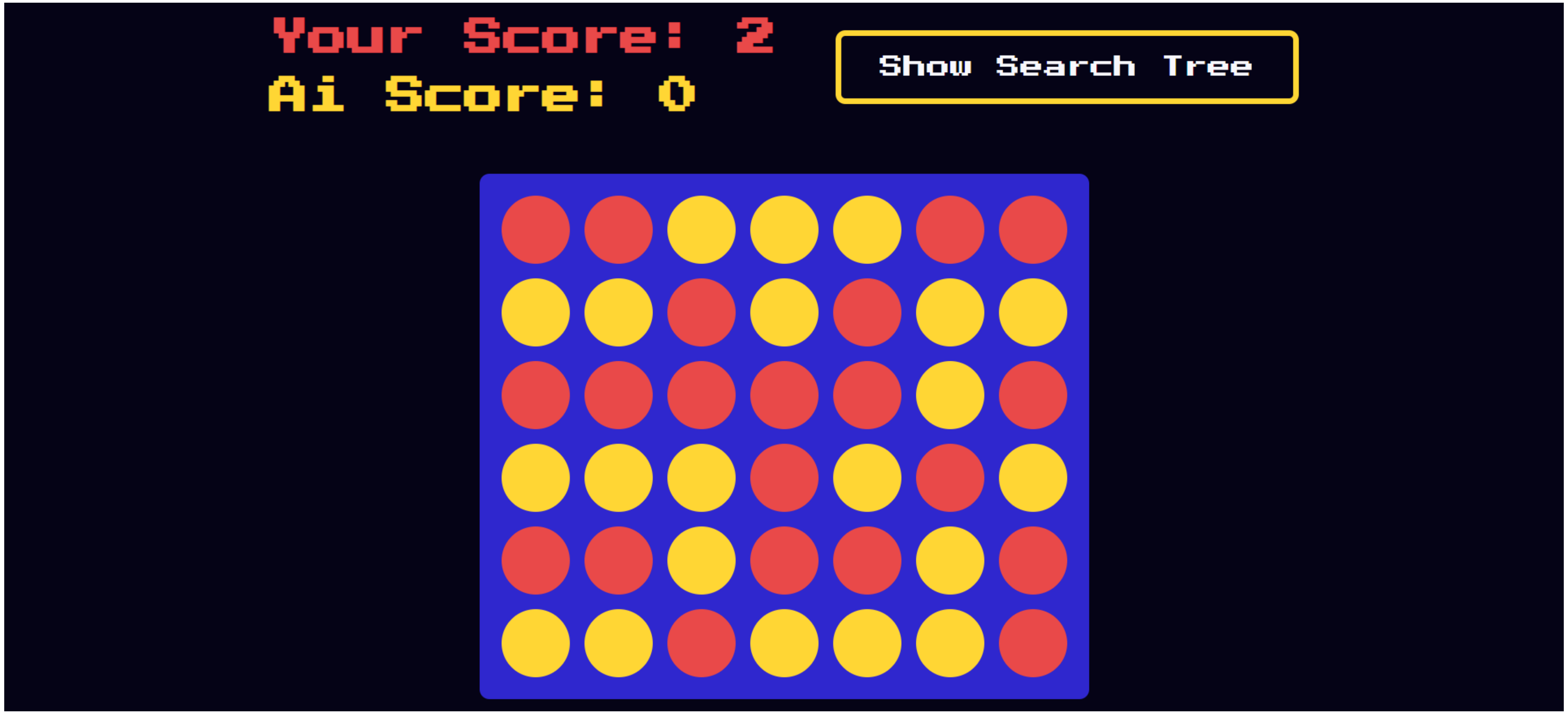
- Y-Axis: Alpha-Beta Time
- X-Axis: Depth



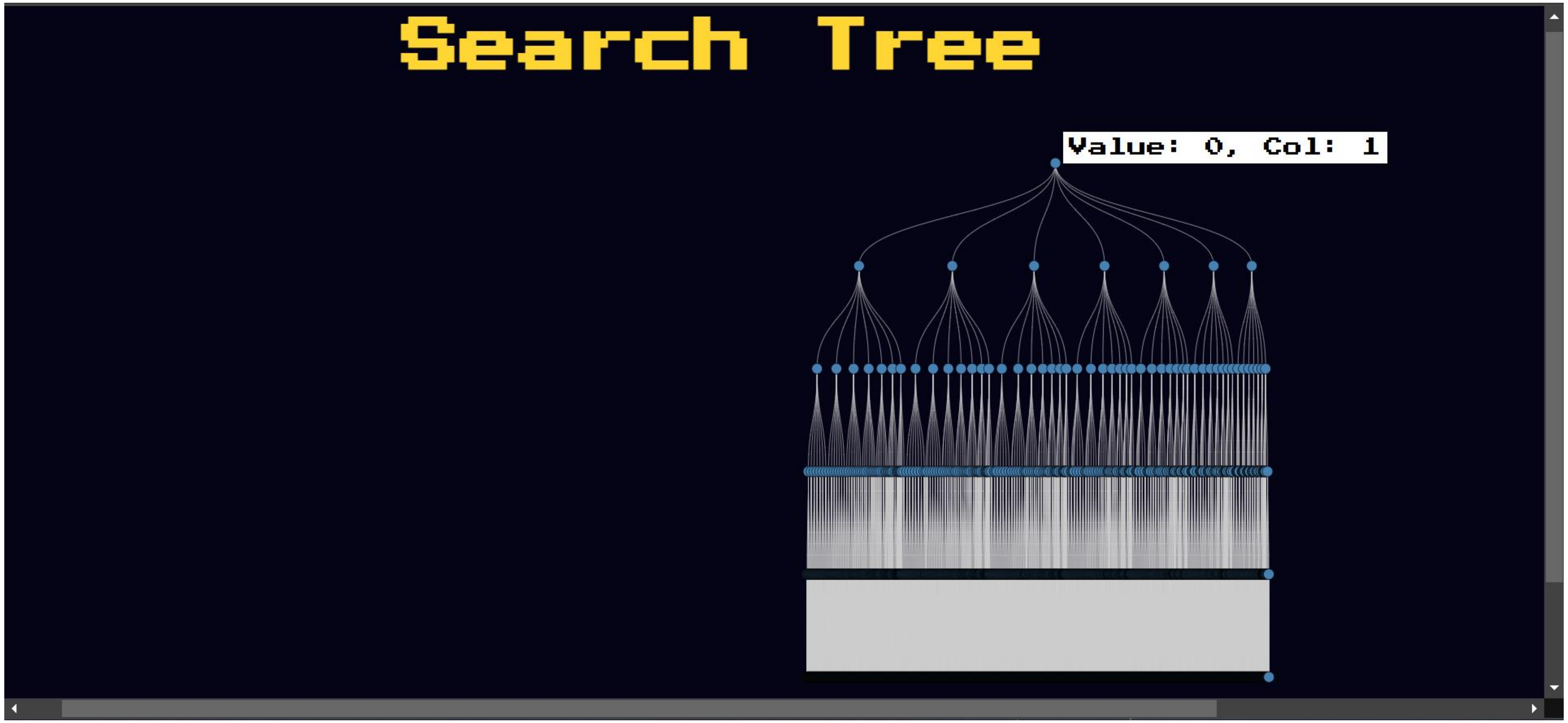
Performance

- Y-Axis: Time
- X-Axis: Depth

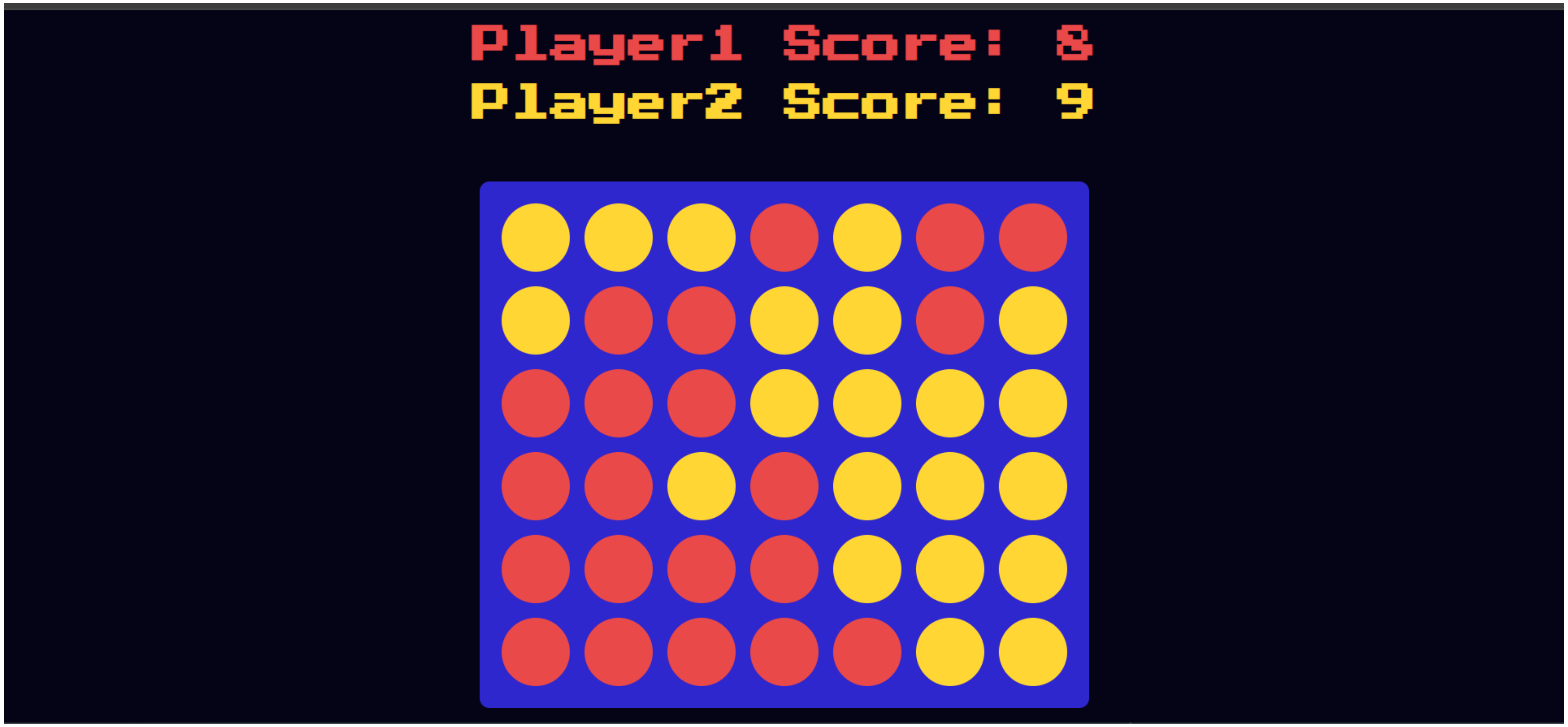
Sample Runs and Tree



Sample Runs and Tree



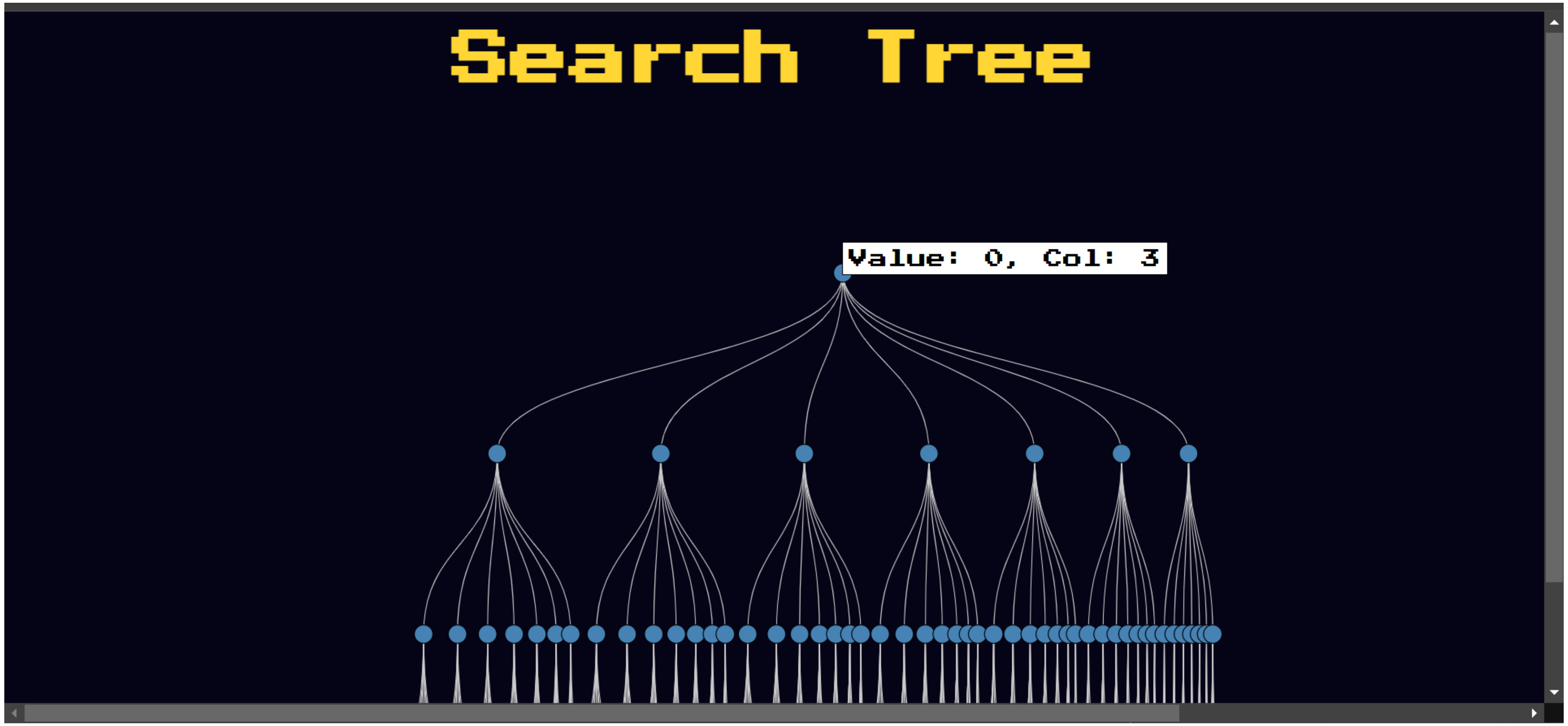
Sample Runs and Tree



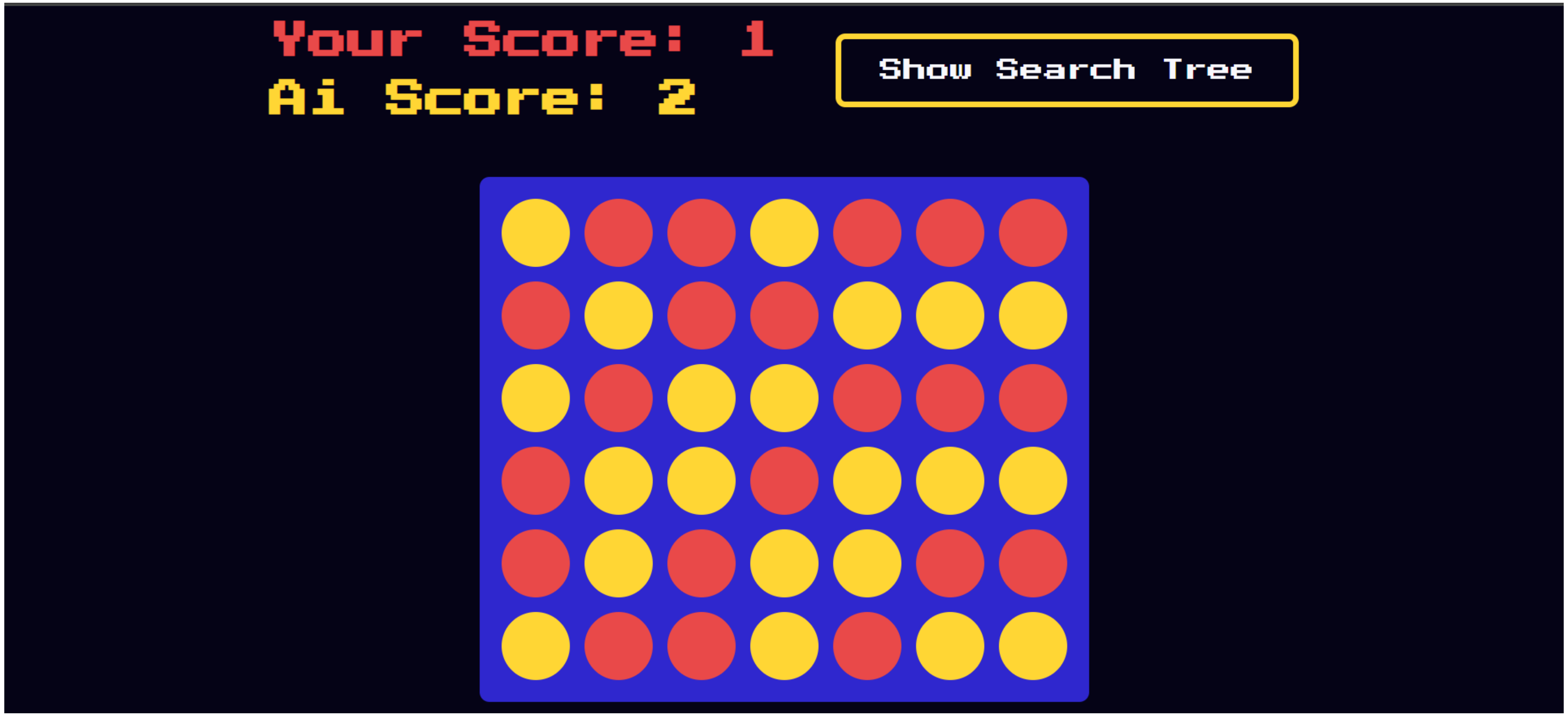
Player1 Score: 8

Player2 Score: 9

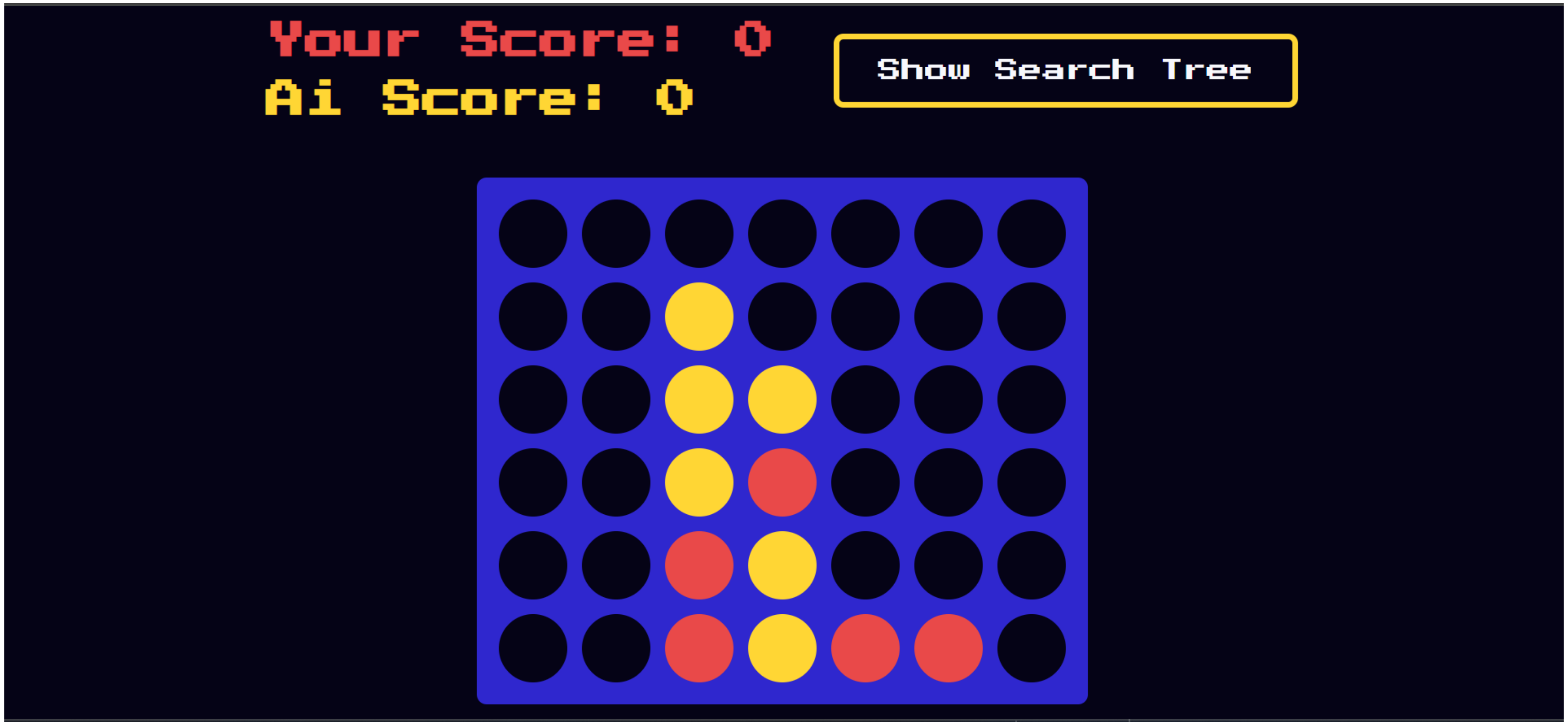
Sample Runs and Tree



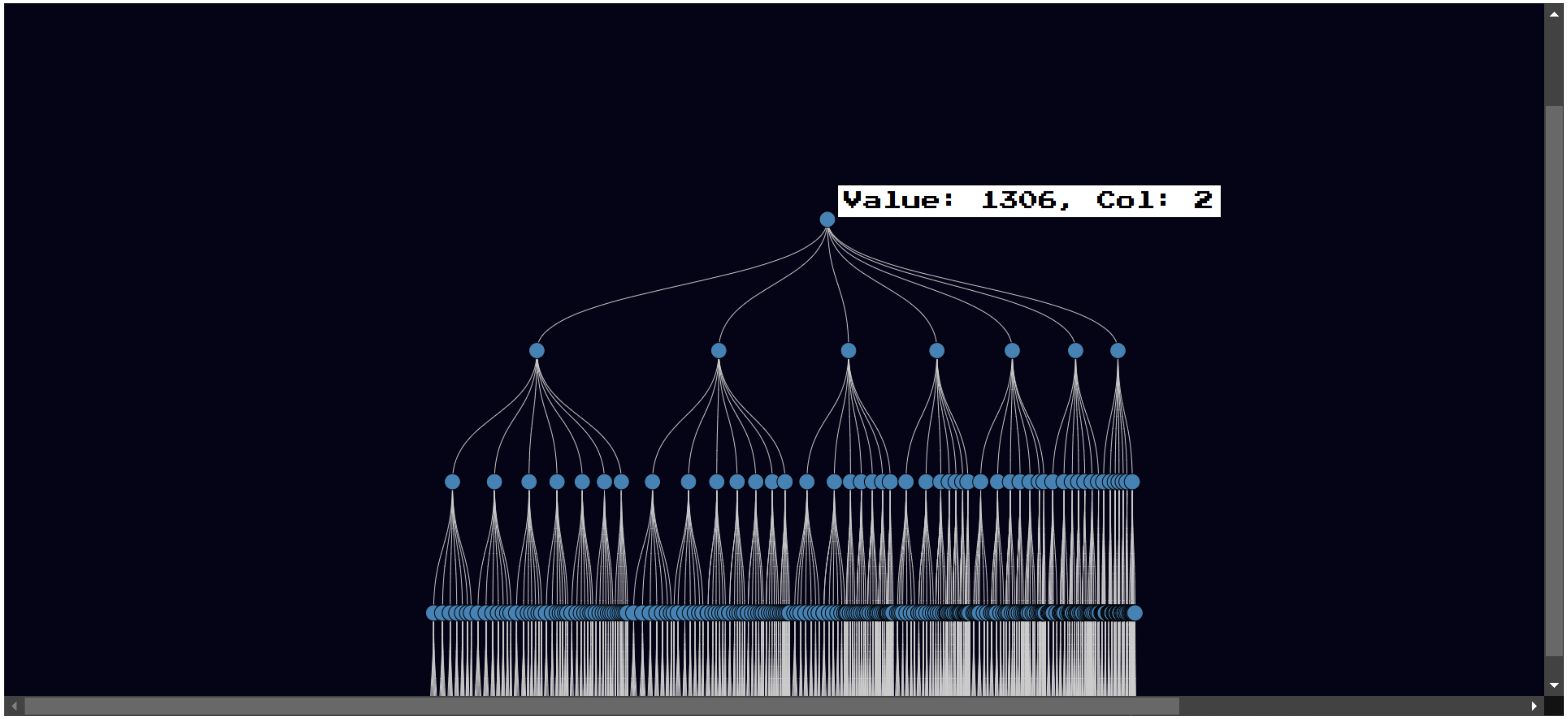
Sample Runs and Tree



Sample Runs and Tree



Sample Runs and Tree



The minimax algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, $\infty$⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility $<$ minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, $-\infty$⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility $>$ maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

$\alpha - \beta$ pruning

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state, $\alpha$, $\beta$)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, $\infty$⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, $\alpha$, $\beta$)

        if utility $<$ minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility $\leq \alpha$:
            break

        if minUtility $< \beta$:
            $\beta$ = minUtility

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state, $\alpha$, $\beta$)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, $-\infty$⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, $\alpha$, $\beta$)

        if utility $>$ maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility $\geq \beta$:
            break

        if maxUtility $> \alpha$:
            $\alpha$ = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state, $-\infty$, $\infty$)

    return child
```

Data Structures Used

- Cache **dict[str, float]** : Stores board states as strings and their corresponding heuristic values for memoization
- **Number of Nodes** : Tracks the number of nodes expanded during the search.
- **List of Children Nodes**